УДК 519.687.7
https://doi.org/10.36906/AP-2020/22

РЕАЛИЗАЦИЯ ПАТТЕРНА «ТРЕХСЛОЙНАЯ АРХИТЕКТУРА» ПРИ НАПИСАНИИ .NET CORE ПРИЛОЖЕНИЯ

Тихонов А. А.
Нижневартовский государственный университет
г. Нижневартовск, Россия
Слива М. В.
канд. пед. наук
Нижневартовский государственный университет
г. Нижневартовск, Россия

Аннотация. В статье рассмотрено использование и реализация паттерна трехслойная архитектура при написании .NET Core приложений. Дано понятие паттернов, описано их предназначение и плюсы использования в современных программных комплексах. Подробно рассмотрены слои приложения и их зоны ответственности.

Ключевые слова: Паттерны, трехслойная архитектура, внедрение зависимостей, dependency injection, бизнес-логика, ASP .NET Core, .NET Core.

Паттерн проектирования — решение определенной, часто встречающейся проблемы при проектировании архитектуры программ. В отличие от готовых библиотек или функций, паттерн невозможно просто откуда-то взять и скопировать в конечную программу. Паттерн представляет собой общую концепцию решения той или иной проблемы, а не какой-то конкретный код, то есть эту концепцию нужно будет подстроить под конкретную программу (https://clck.ru/T7iAg).

Паттерны часто сравнивают с алгоритмами действий, так как оба этих понятия описывают типовые решения каких-то известных, наиболее часто встречающихся проблем, но это не совсем соответствует действительности. Алгоритм — четкий набор действий, в то время как паттерн — это высокоуровневое описание решения, без описания деталей, реализация которого может отличаться от решения к решению. Если привести аналогии, то алгоритм — это кулинарный рецепт с четкими шагами, а паттерн — инженерный чертеж, на котором нарисовано решение, но не конкретные шаги его реализации. Зачем же программистам нужно использовать паттерны (https://clck.ru/T7iAg)?

— Паттерны — проверенные решения. Меньшая трата времени с помощью готовых решений, вместо изобретения велосипедов.

— Стандартизация кода в приложении. Программист делает меньше просчетов при проектировании, используя типовые унифицированные решения, так как все скрытые проблемы при их использовании уже давно найдены.

— Общий программистский словарь. Вы произносите название паттерна, вместо того, чтобы час объяснять другим программистам, какой крутой дизайн или архитектуру вы придумали, а также какие классы для этого нужны.

— Облегчение сопровождения программного обеспечения, так как все программисты знают наиболее известные паттерны.

Трехслойная архитектура — архитектурная модель приложения (рис. 1), которая предполагает наличие в нем трех слоев: клиента, сервера приложений и сервера баз данных (https://clck.ru/T7i8G).

Компоненты трехслойной архитектуры:

Слой представления (клиент) — интерфейс приложения, предоставляемый непосредственно пользователю. Данный слой, согласно паттерну, не имеет прямых связей с хранилищем данных, чтобы удовлетворять требованиям безопасности, а также критерию масштабируемости, не быть нагруженным бизнес-логикой приложения (по требованиям масштабируемости) и хранить состояние приложения (по требованиям надежности). На этот слой помещается только самая простейшая логика: интерфейс авторизации, валидация вводимых данных, а также операции с данными, которые уже загружены, например сортировка, группировка или подсчет значений.

Слой сервера приложений (связующий слой, бизнес-логика) располагается на втором уровне, на нем должна располагаться основная бизнес-логика приложения. Вне его остаются только фрагменты, экспортируемые на клиента (терминалы), а также элементы логики, погруженные в базу данных (хранимые процедуры и триггеры). Реализация данного компонента обеспечивается связующим программным обеспечением.

Сервер баз данных (слой данных) обеспечивает хранение данных и выносится на отдельный уровень, реализуется, как правило, средствами СУБД, подключение к данному слою происходит только с уровня бизнес-логики приложений.

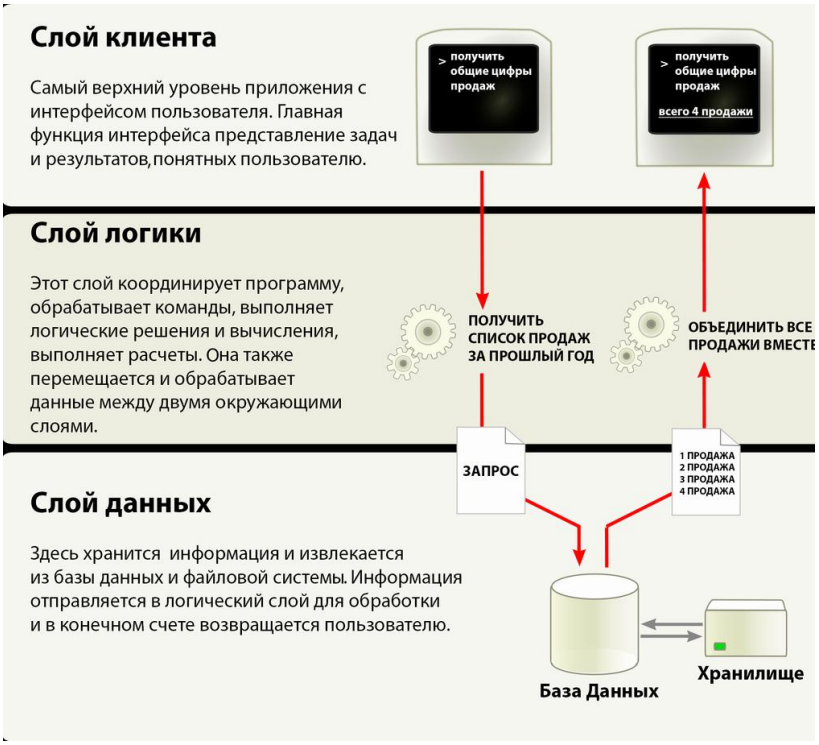

Рис. 1. Схема трехслойной архитектуры.

В простейших конфигурациях, например в домашнем проекте, все компоненты или часть из них могут располагаться на одном и том же вычислительном сервере. В конфигурациях, которые используются на реальных проектах, как правило, используется выделенный вычислительный узел для сервера баз данных или несколько совмещенных серверов баз данных, для серверов приложений — выделенная группа вычислительных узлов, к которым непосредственно подключаются клиенты (терминалы).

Трехслойная архитектура реализуется на основе слабого связывания и внедрения зависимостей с помощью специальных фрейморков. Для реализации трехслойной архитектуры на языке C# и платформе .NET Core нам необходимо создать несколько сборок, которые в данном случае будут соответствовать слоям нашего приложения, а также сборки с контрактами (необходимыми интерфейсами) для слабого связывания (рис. 2).

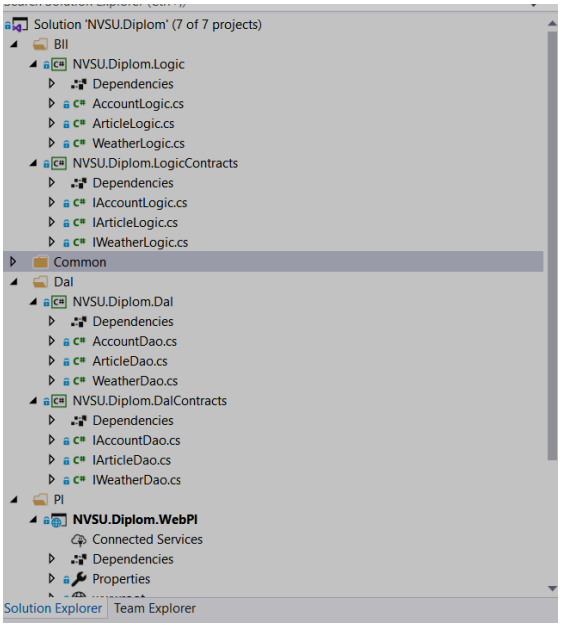

Рис. 2. Скриншот обозревателя решения в IDE Visual Studio.

В итоге в приложении имеются следующие сборки:

— NVSU.Diplom.Entites — сборка с сущностями приложения;

— NVSU.Diplom.Logic — сборка с классами, отвечающая за слой бизнес-логики приложения;

— NVSU.Diplom.LogicContracts — сборка с интерфейсами, отвечающая за контракты бизнес-логики приложения;

— NVSU.Diplom.Dal — сборка с классами, отвечающая за слой общения с хранилищем данных;

— NVSU.Diplom.DalContracts — сборка с интерфейсами, отвечающая за контракты хранилища данных;

— NVSU.Diplom.WebPl — сборка, отвечающая за слой представления данных, реализующая технологию ASP .NET Core MVC, здесь находятся классы, отвечающие за общение с пользователем программного комплекса.

Каждая сборка отвечает за определенные действия и соответствует принципу единственной ответственности, одному из принципов объектно-ориентированного программирования и проектирования SOLID.

— S: Single Responsibility Principle (Принцип единственной ответственности).

— O: Open-Closed Principle (Принцип открытости-закрытости).

— L: Liskov Substitution Principle (Принцип подстановки Барбары Лисков).

— I: Interface Segregation Principle (Принцип разделения интерфейса).

— D: Dependency Inversion Principle (Принцип инверсии зависимостей).

Внедрение зависимостей

В программной инженерии внедрение зависимостей — это метод, при котором объект получает другие объекты, от которых он зависит. Эти другие объекты называются зависимостями. В типичных отношениях «using» принимающий объект называется

клиентом, а переданный (то есть «внедренный») объект – службой. Код, который передает службу клиенту, может быть разного рода и называется инжектором. Вместо того, чтобы клиент указывал, какую службу он будет использовать, инжектор сообщает клиенту, какую службу использовать. «Внедрение» относится к передаче зависимости (службы) объекту (клиенту), который будет ее использовать.

Работа специальной основы (англ. Framework), обеспечивающая внедрение зависимости, описывается следующим образом. Приложение, независимо от имплементации, исполняется внутри контейнера «инверсии управления», предоставляемого специальной основой. Часть объектов в программе по-прежнему инициализируется обычным способом языка программирования, а часть создается контейнером на основе предоставленной ему конфигурации.

Условно, если объекту нужно получить доступ к определенному DataSource, объект берет на себя ответственность за доступ к этому DataSource: он или получает прямую ссылку на местонахождение DataSource, или обращается к известному «DataSource-локатору» и запрашивает ссылку на реализацию определенного типа сервиса. Используя же внедрение зависимости, объект просто предоставляет свойство, которое в состоянии хранить ссылку на нужный тип сервиса; и когда объект создается, ссылка на реализацию нужного типа сервиса автоматически вставляется в это свойство (поле), используя средства среды.

Внедрение зависимости более гибко, потому что становится легче создавать альтернативные реализации данного типа сервиса, а потом указывать, какая именно реализация должна быть использована в, например, конфигурационном файле, без изменений в объектах, которые этот сервис используют. Это особенно полезно в юнит-тестировании, потому что вставить реализацию «заглушки» сервиса в тестируемый объект очень просто.

В ASP .NET Core есть встроенный контейнер для внедрения зависимостей, что очень удобно, так как не нужно устанавливать никаких дополнений, как было в предыдущих версиях, например в ASP.NET 4 надо было использовать различные внешние компоненты, такие как Windsor, Ninject, Castle, StructureMap, Autofac, Unity, хотя при желании их также можно продолжать использовать и в новой версии.

Управление зависимостями осуществляется в классе Startup.cs, сборки слоя представления данных (рис. 3).

```
1 reference
private void ConfigureDependecies(IServiceCollection services)
{


    ConfigureWeather(services);

    services.AddTransient<IAccountLogic, AccountLogic>();
    services.AddTransient<IArticleLogic, ArticleLogic>();

    services.AddTransient<IAccountDao, AccountDao>(serviceProvider =>
    {
        return new AccountDao(connectionString);
    });
    services.AddTransient<IArticleDao, ArticleDao>(serviceProvider =>
    {
        return new ArticleDao(connectionString);
    });
}
```

Рис. 3. Внедрение зависимостей в классе Startup.cs.

Ключевые понятия встроенного контейнера:

IServiceCollection, это службы регистрации, которые сообщают контейнеру о конкретной реализации интерфейса. Его следует использовать для определения того, какой интерфейс какой реализации принадлежит. Создание чего-либо может быть таким же простым, как создание экземпляра объекта, но иногда нам нужно больше данных.

IServiceProvider, разрешает экземпляры службы, собственно смотрит, какой интерфейс какой конкретной реализации принадлежит и выполняет создание.

Он находится в пространстве имен Microsoft.Extensions.DependencyInjection. Здесь можно найти все классы и код, относящиеся к DependencyInjection.

Таким образом, при грамотном использовании паттерна и встроенных возможностей платформы .NET Core можно создавать высоконагруженные, отказоустойчивые, легко масштабируемые программные комплексы, отвечающие современным требованиям к программному обеспечению.